periodica polytechnica

Social and Management Sciences
14/1 (2006) 19–28
doi: 10.3311/pp.so.2006-1.03
web: http://www.pp.bme.hu/so

RESEARCH ARTICLE

Software development as social activity: distributed cognition or hermeneutic pratice?

Viktor Binzberger

Received 2007-09-14

Abstract

How shall we render understandable the social-practical processes of software development? First, I am going to present the theory of Distributed Cognition (dCog), because, given its privileged position within Human-Computer-Interaction (HCI) literature, this is the most likely candidate for a philosophical theory of software development.

Next, I am going to demonstrate with a case study that the processes, which I call hermeneutic activities, lie outside the domain of this theory. These hermeneutic episodes are characterized exactly by the lack of a commonly shared functional description level, but the existence of such a level is indispensable for the theses of dCog to hold. According to my argumentation, the hidden premise that assumes the existence of such a level is not only problematic, but is also inconsistent with the other theoretical roots of dCog. In order to see this, we have to turn our attention toward the practices of interpretation that are taking place in situated hermeneutic activities.

In my analysis, I am going to lean on the other branch of dCog's theoretical roots, predominantly on the works of Suchman, Winograd, Dreyfus, and Norman.

Keywords

software development · distributed cognition · hermeneutics

Acknowledgement

I am thankful for the support of the BME-HAS Bela Julesz Cognitive Science Research Group, and the Jedlik Ányos Research Grant (NKTH KPI NKFP6-00107/2005).

Viktor Binzberger

Department of Philosophy, BME, H–1521 Budapest, Műegyetem rkp. 3., Hungary

e-mail: bviktor@filozofia.bme.hu

1 Introduction

How shall we render understandable the social-practical processes of software development? Beside the non-systematic, much-debated, yet influential literature of the practitioners' self-reflection (e.g. Beck, 1999; Hunt, 1999; McBreen, 2002; Boehm, 2004), there are only few systematic investigations that are based on classical *philosophical* approaches. Given the widespread acknowledgement of the societal relevance of information technologies, it seems strange that philosophical theories do not pay enough attention to the *particular social-practical processes* in which contemporary info-communication technologies *come into being*.

In the following, I would like to present the theory of distributed cognition (dCog), because, given its privileged position within HCI literature, this is the most likely candidate for a philosophical theory of the social processes of software development. Its most important intellectual ancestor is Simon's and Newell's influential theory of symbol-processing and rational decision-making, but it also departs from it in significant ways, drawing upon theorists like Brooks, Suchman, Winograd, Dreyfus, Maturana, Lakoff and Norman, who all criticize the rationalistic, disembodied approaches of cognition and computing: [4, 11, 29, 31, 35, 39, 43]

I am going to demonstrate with a case study that the processes, which I call *hermeneutic activities,* lie outside the domain of dCog. These hermeneutic episodes are characterized exactly by the *lack* of a *commonly shared functional description level*, but the existence of such a level is indispensable for the theses of dCog to hold. According to my argumentation, the hidden premise that assumes the existence of such a level is not only problematic, but is also inconsistent with the other theoretical roots of dCog. In order to see this, we have to turn our attention toward the practices of *interpretation* that are taking place in situated hermeneutic activities, and we have to handle concepts like "representation" and "information" with suspicion, because they tacitly imply the existence of such a shared functional description level.

In my analysis, I am going to lean on the other branch of dCog's theoretical roots, predominantly on the works of Such-

man, Winograd, Dreyfus, and Norman. I'd like to emphasize and provide support for the following theses:

a) Understanding unfolds itself in *skillful action*. This means that understanding, in its most originary sense, does not mean holding and manipulating symbols in our "mind", but rather the capability of orienting ourselves – coping – within a life-world. (Dreyfus 1998) Using language to orient ourselves mutually in a shared practical engagement is of special importance, but is still skillful action.[1]

b) Understanding is *situated* and *embodied*, that is, always already embedded into the material concreteness and contingencies of the situation [35, 39].When in giving account of understanding, we abstract away the idiosyncrasies of the situation, we risk losing a central feature of understanding.

c) Understanding presupposes having already taken up a position within *a holistic horizon of meaning, constituted by a shared lifeworld and shared traditions of interpretation*. In the process of understanding, this horizon gets reinterpreted and transformed to encompass new perspectives on new domains.

d) Understanding – even as skillful action – always involves an *ongoing interpretation* of the surrounding world. The everyday routine of abstracting away from the embodied situation to describe it within a linguistic conceptual framework is just one extreme example of this. Skillful action without words already involves perceiving the world as purposeful, as one that offers affordances, as one that can be translated into *a field of possibilities for action and existence* [17].

Through criticizing dCog, I do not intend to prove its untenability. As we will see, dCog is compatible with my theses a), b) and c): my goal is to extend and supplement it with thesis d).

2 Distributed Cognition

The term Distributed Cognition was coined by Edwin Hutchins [23].The main tenets of this theory are summarized in [17]:

i) A cognitive process is delimited by the functional relationships among the elements that participate in it, rather than by the spatial collocation of the elements;

ii) Whereas traditional views look for cognitive events in the manipulation of symbols inside individual actors, distributed cognition looks for a broader class of cognitive events and does not expect all such events to be encompassed by the skin or skull of an individual [19].

and these lead to the following theses:

1 Cognitive processes may be distributed across the members of a social group.

2 Cognitive processes may involve coordination between internal and external (material or environmental) structure.

3 Processes may be distributed through time in such a way that the products of earlier events can transform the nature of later events [19].

An interesting application of the theory is the case study of Hutchins & Palen (1997), in which they investigate the interaction between the pilots in an emergency condition during a simulated flight. They engage themselves in a distributed troubleshooting process, systematically searching through the space of possible causes of the instrumental readout. The authors notice, that in conveying meaning, the pilots rely extensively on their shared perceptual access to the instrument panel and on their shared skillful knowledge about possible courses of action. The spatial distribution of the instruments plays a significant role in visualizing inferential steps of the troubleshooting process. Following the gestures and the explanation of the second officer, the pilots shift between different levels of representation, sometimes looking at the instrument as a fuel gauge and then taking it to be a representation of the fuel tank itself. In the first case, they look at the fuel panel, and in the second, they "see through" it. When the secondary officer silently puts his finger onto the fuel test switch, they correctly infer that he means that he has already tested the fuel panel hardware – this shifts the level of representation to the meta-level, involving inferences about each other's knowledge and responsibilities. The authors arrive at the conclusion that articulated speech is just one modality among others: "space, gesture, and speech are all combined in the construction of complex multilayered representations in which no single layer is complete or coherent by itself."

2.1 The Classical Cognitivist Roots of Distributed Cognition

The approach of the dCog theorists draws upon the classical cognitivist ideas of Simon and Newell (1976), and they take them further in certain directions. If cognition is *symbol manipulation* and its motivation is *problem solving*, as cognitivists say, why should cognition be narrowed down either to the confines of the individual brain, or to the housing of the "intelligent" computer? Symbol-manipulation processes of problem solving are distributed among people and their artifacts. In Hutchins' example, touching the fuel panel test switch externalizes the cognitive process of the second officer: by sharing his belief that the problem does not reside in the fuel panel, he effectively cuts down

[1] In cognitive psychology, Michael Tomasello's theory of language acquisition stands very close to my approach. His concept of "shared attention", and his emphasis on the importance of coordinated action in learning a language is in accord with theses a) and b). The most significant difference is that Tomasello sees continuity between classical cognitivism and his situated model: for example, he uses concepts like "plan" and "plan recognition", whereas theorists whom I build upon, like Dreyfus and Suchman, argue that there is a significant discontinuity between situated and classical symbolic theories of cognition, and the concept of "plan" is a misleading and obsolete metaphor (Tomasello 2005; Suchman 1987).

the dead-ends of the search for solutions in the problem space, thus eliminates redundancies in this parallelized problem solving effort.

The fingerprint of classical cognitivism can also be discerned from other aspects of the example. In Hutchins' description, the features of space, gesture, and speech are already presented as symbolic *representations*, preselected, segmented, and categorized according to their cognitive role in solving the problem. Lines of explanation along other social aspects are not taken into account: knowing that it is a videotaped simulation, it might well be, that the second officer touches the test switch with the intent of *delegating responsibility*, saying, in effect "I have followed the standard textbook procedures and now I am awaiting further orders." Emotions, social status, power struggles do not play a role in these descriptions. DCog's other examples – ship navigation (Carroll 2003), informal techniques of using the airspeed indicator, the use of the cursor and the "airstrip" in air traffic control [10, 14, 27] are all characterized by a narrow cognitivist focus on symbolic or symbolically reconstructable interactions. All their examples consider already well-established, *routine* practices, with a preexisting ontology of action discernible from rulebooks and technical literature. Their description is not a phenomenology of action in a Merleau-Pontian sense, but is rather a functional description of a meta-individual machinery, within which humans play a mechanistic role.

Their approach recently became attacked as one that does not make sufficient distinction between humans and machines. The debate with their answers is summarized in [40], but it seems to me, that since dCog theorists share their premises with Simon and Newell, on which the classical argument for the possibility of artificial intelligence is built, they must commit themselves to its conclusions also.

2.2 The New Waves of Cognition: Embodiment, Situatedness, Ethnomethodology

On the other hand, dCog theorists draw on many ideas from Brooks, Clark, Lakoff, Norman, Suchman, Varela and Winograd, who are all very critical toward classical cognitivism. They embrace the idea of *embodiment* by emphasizing the importance of the particular material realization of instruments and administrative artifacts in our cognitive processes. The airspeed meter with an analog gauge is better than a digital one, because it can be used for perceptual reasoning about relative speeds, and cardinal directions are a more convenient way to communicate speeds than decimal numbers [24]. The paper file can accommodate such extra information (handwriting, post-it notes, etc.) in an ad-hoc fashion that the badly designed computer database cannot. They also reflect on the *organization of space* as an influental heuristic tool for thinking: e.g., the placement and grouping of computer icons on the virtual space of the screen can encode important meta-information about the files they represent. But their concept of embodiment is still a narrowly cognitivist one: they still take the material realization of tools into account only so far as they exhibit representational capabilities.

They follow Suchman in describing action as *situated* within an unarticulated material background, which serves as a shared resource, upon which all participants of a discourse must draw in order to understand each other. Reasoning, deixis, and disambiguation of words take into account visual, spatial, and cultural metaphors. Their view also endorses Norman's argument, according to which intelligence is not "in the head", but rather in the cultural heritage of the well-designed artifacts, which surround us, and "make us smart", and in the processes in which these artifacts come into being.

The tacit dimension of knowledge or knowledge as skillful action – Winograd's argument against classical representationist cognitivism – is present in the theory, but it does not play a central role.

They also use an *ethnomethodological* approach, which is quite different from anthropological research on indigenous people, since they employ the ontology of their subjects in their descriptions ("actor's categories"), but this ontology is far from being naive: it is thoroughly informed by the legacy of 20^{th}-century administrative science and operation research.

So far there is no disagreement between dCog and my approach. The above aspects are covered by my theses a), b) and c).

2.3 My critique against dCog

There is an internal contradiction between the functionalism dCog theorists are committed to in principle i), and the emphasis on embodiment and situatedness. If we apply the arguments of Norman, Suchman, Varela, etc. to the concepts of the *analyst*, we find that they also have to be situated and embodied, drawing upon a shared material background, and so on. On the other hand, the universal functionalism of i) would require an *independent* level of functional description, a level that is applicable equally to all participants of the distributed cognitive process, humans and non-humans alike. Such a functional level presupposes a more or less detailed ontology of action (how the seamless flow of action can be segmented and categorized conceptually into discrete acts) and an ontology of the problem (what are the features that are relevant in finding a solution for the problem, what kind of decision alternatives and computational states can be identified). To build such a functional description, we either have to adopt the concepts of the participants (as Hollan et al. in fact do), or we have to get involved in the situation. Now, if the descriptions clash with each other, whose ontology shall we prefer?

When dCog theorists talk about multimodally transmitted representations, materially embodied information and so on, these things "in themselves" are nowhere to be found in the flow of action. We can only talk about material processes as instantiating a symbol-transmission when we take the stance of an interpreter, and conceptualize and arrange the action into a functional narrative (a "plan") [39]. Talking about "symbols"

and "functions" presupposes an act of *interpretation*.

When building a theory of distributed cognition, we have to keep in mind that symbols and functions are not part of a reality which is independent from the participants and the analyst, but rather the participants and the analyst are those, who delineate them and give them meaning. When the horizon of interpretation is standard between participants and analysts – and this is the case in the standard dCog examples, which all focus on well-defined, routine processes – this act of interpretation is generally not reflected upon. But when this horizon is not standard – the use of symbols and their meanings differ among participants –, we cannot assume the existence of a common functional description level. In order to build out this shared level of description, within which they can identify symbols and functions, participants, and analysts have to engage in interpretative activity. *It is exactly this kind of interpretative activity, which falls outside the domain of dCog, because here there are no fixed set of symbols and functions yet*; for solving the problem, the alternative branches of the search tree, the possible categorizations of actions and computational states are yet to be delineated. Not symbols or representations are transmitted in these interpretational discussions, but the perspective itself, which makes possible the representation of the problem domain and the symbolic definition of possible actions towards its solution.

In the following, I would like to present the case study, with which I would like to show that a great part of problem solving consists exactly of this kind of interpretative activity. Theorists as Norman, Suchman and Winograd are the ones on whom we can build upon, if we want to reach a deeper understanding of this practice.

3 Case study: Understanding the Source Code – The Role of Abstraction

3.1 Introduction

Software systems and their developers, viewed as a distributed cognitive system, working together to build a solution to a problem domain, seem to fulfill principle i) of dCog, since there is a given common level of functional description that is shared by developers and computers alike: this is the source code. The programming language has a standard interpretation that maps source code onto the physical operations carried out by the computer unambiguously, and this interpretation is embodied in the compiler/interpreter program. Yet still, the breakdown of understanding between computer programmers is one of the most feared phenomena among programming language designers [2]. Do programmers really understand source code as a functional description of the problem at hand? Do the symbols of the source code mean the same for all participants? What kind of meta-level understanding do they bring into play when they reflect on and rework the source code?

I am going to show here that the source code comes into being in a process of interpretative activity, and it is subject to further reinterpretation and revision. The current state of the source code, the "development snapshot" is only understandable within a practical interpretative tradition that is constantly rebuilding its own horizon of understanding by improving its conceptual tools and its instrumental environment, and the kind of understanding achieved by them is best characterized in the light of the theses a) – d).

There are at least three core features of software development practice, which are central to our investigation.

3.1.1 Abstraction

is a loose set of practices, through which the particular software solution gets disentangled from the concreteness and contingencies of the use-situation, so that a multiplicity of use-cases gets thinkable and controllable with a finite set of symbolic representations. Abstraction can also be thought of as an "imaginary induction" over the space of possible future use-cases, while looking for lawlike regularities in the projected patterns of use. Modularization, functional decomposition and metaprogramming are all examples of abstraction, supported by certain programming languages. The importance of abstraction lies in that it reduces the holistic tangle of relationships of the system and of the problem domain to be thinkable and communicable among developers, sometimes with the explicit aim of "eliminating" their holism. Abstraction makes it easier for the non-initiated developer to situate himself in the background necessary to understand a portion of source code.

3.1.2 Embeddedness within a tradition of interpretation

refers to the phenomenon that preexisting abstractions of the problem domain – residing in traditional mathematical formulations or natural-language conceptualizations – provide traditional horizons of understanding, which can be relied upon during building a new system. In the case of *metaprogramming*, for example, the goal is to build up a translation between a traditional conceptualization of the problem and the abstractions provided by the system; between the problem-specific language and the system-specific computational tools [17].

Embeddedness in a tradition does not necessarily mean conservativism. The metaphorical base of a traditional language can undergo great shifts to accommodate new perspectives. Peter Galison describes in a wonderful article how the very first instance of computer use by John von Neumann to simulate the hydrogen bomb explosion introduced such a shift of traditions [13]. Given the complexity of the simulation, Neumann could not have solved his problem with the traditional approach, by the way of symbolically solving differential equations. Having access only to the very limited computational capacities of the ENIAC, Neumann had to find a way to reduce the number of computational steps necessary to calculate the equations analytically. He came up with the idea of random sampling, borrowed from his earlier explorations into game theory, which later developed into the so-called Monte-Carlo method of simulation. Furthermore, this idea also had the effect of displacing the tradi-

tional conceptualization of hydrodynamic problems in terms of symbolically solvable differential equations with a new conceptual tradition, one that laid emphasis on the inherent stochastic properties of atomic processes, and was easier to translate into the language of the available computational tools.

3.1.3 Standardization and extending of controlled microworlds

. Every abstraction that reduces the complexity of the system achieves this aim by grouping together different use-cases under the same symbolic articulation. This reduces the total diversity of possible situations that might appear in the problem domain by levelling down the differences between them in certain dimensions and accentuating them in others. This "levelling down" can take the form of an adaptation to the structure of the problem domain as it is previously given within a traditional horizon, or it can also play the role of a structuring force that reorganizes the problem domain by introducing similarities and augmenting differences that are traditionally not to be found there. Instead of conforming to the world, this latter form of abstraction makes the world conform to the system, for example, by enforcing standardized ways of interacting with it or by declaring strictly the possible uses of its communication interfaces. To elucidate this point, I am going to borrow Rouse's concept of "microworld", which is especially relevant here, given its rootedness in AI research [37, 38, 43] . Rouse argues that the success of science does not lie in being able to model and predict everything, but rather in being able to construct and *extend microworlds*. Microworlds are reduced models, laboratory settings with finite number of variables that can be controlled and can have their relationships thoroughly explored. In AI research, microworlds are simulated environments in which AI algorithms can be fully tested. The key to success lies in finding a balance between adapting the algorithm or the scientific product to all contingencies of the real world, and between transforming the world itself to bring it into alignment with the conditions presupposed by the algorithm or the product. This is rather well demonstrated by Hounshell's study (1992) [20], who examines two episodes from the history of Du Pont, where this "scaling up" did indeed succeed: the story of the nylon, and the case of the Hanford nuclear plant (the development of the first atomic bomb). In the case of the nylon, the developers faced the task of turning an instable, water-soluble, hard-to-prepare laboratory sample into a stable, water-resistant, mass-producible material, and simultaneously, adapt to existing market demands (women stockings), then leverage this market position to transform and channel various other markets into Du Pont customers. In the case of the Hanford plant, they had to drive up plutonium production from the microgram scale of the laboratory up to the kilograms needed for the bomb. They had to extend the microworld of the laboratory into a plant that at times employed as much as 60'000 people, and simultaneously, they had to stabilize the vast, heterogeneous network of military decision-makers, scientists and technologists by finding ways to align their interests with those of the big project.

To pick out an example of alignment from IT, for a long time it proved much easier to educate people to write standard characters on a touch-screen display, to make them align to the system, instead of devising an algorithm that recognizes or learns various types of handwriting. Adapting the software to the fuzzy variability of the world might address more users, but it also makes it more complex, since it breaks down the commonalities that could be grasped with the same abstraction. On the other hand, standardizing the possible interactions with the system opens up the opportunity of introducing higher abstractions.

From this perspective, following Rouse's argument, abstraction can also bee seen as having an important dimension of *power*. The central question is; how can a handful of developers impose their own abstractions, their own dimensions of similarity and difference on the multiplicity of their users and their use-cases? *How can they extend their microworld into the lifeworld of their users?*

3.2 The Failure of Understanding

The case study is about a market-leader service provider that offers its services worldwide through the internet[2]. My interviewee, John has the task of understanding the source code of the database-management system (DBM) that the senior programmer Anthony has written, in order to be able to improve and modify it. John was appointed to this job because Anthony had a rolling backlog of maintenance tasks and could not devote any time to implement new features, and because he started to become a critical risk factor: he alone was able to maintain and develop the DBM. This had an effect also on the firm's market value: a firm, whose critical infrastructure can only be maintained by one particular person, is not worth much in the eyes of potential investors. John's role is thus twofold: reducing the performance bottleneck and the risk factor Anthony poses.

After a short while, John started to perceive the task as nearly unaccomplishable. The DBM consisted of 4 megabytes of C++ source code, without any documentation and very scarce source code comments. This in itself would not necessarily pose a problem, since there are plenty of such large and badly documented systems (the Linux kernel being one of them), that are intensely developed by a highly decentralized group of developers. But in this case, the source code resembles a cryptogram. Variable- and function names are meaningless abbreviations („a", „p", „prqi", etc.). There are no coding conventions; the source code lines are extremely long. The source code contains lots of repeating patterns. The code is highly redundant: there are at least fifty implementations of the quicksort algorithm, which all differ only in some minor details. The source code contains many branching points[3] at which the code forks into alternative, unique so-

[2] The names of the interviewees have been changed to protect sensitive information.

[3] It is, in fact, a complex, ad-hoc system of embedded code fragments imple-

lutions for each server[4] and service subscriber. All individual settings are hardwired into the source code; there are no configuration files. The abstractions supported by the C++ language are totally absent. There are no objects, templates and namespaces, and there are only a very limited number of function calls. Functions with thousand-line bodies are not rare.

Soon it also turned out, that the situation involves a certain psychological dimension. *Anthony cannot articulate and explain the working of the system.* It is not that Anthony would not know how it works. The system works quite efficiently and reliably, it uses many complex solutions (an own file system, for example). Anthony can carry out the maintenance and development tasks alone, and he can also explain the working of any particular code snippet step-by-step, but he cannot explain the working of the system *in general*. He is aware that the system is incomprehensible to anyone except himself, but he argues that the complexity arises from the system being painstakingly optimized and customized to best suit its customers' needs: complexity is a tradeoff for appropriateness and efficiency. But this statement does not fully cohere with reality: the system has reached its total capacity, the firm cannot sign contracts with new subscribers. Anthony is exhausted and no other programmer can help him.

It is also interesting that the source code is almost fully independent; it does not rely on any other source code or function library. According to Anthony, "you can only trust what you've coded yourself". Code written by others is incomprehensible, cannot be thoroughly understood, so it cannot be trusted.

According to John's description, the code does not abstract away from the level of the source code: it is actually machine-level (assembly) code written in C syntax. "Anthony doesn't trust code, behind which he can't see the assembly", as John notices. Anthony even checks the code produced by the compiler in the disassembly window. This might improve run-time efficiency, but it has a drastic effect on developer-time efficiency. The individual, idiosyncratic solutions make it impossible to gather similar use situations under a shared abstraction. The improvement on run-time is also questionable. All the fifty quicksort routines are highly customized, but none of them is extremely efficient, because they all implement a naive quicksort algorithm. Instead of fifty quicksort routines, it would be better if there were only a single one that were carefully crafted, well tested, and optimized.

But why is it so impossible to understand this source code? We have an all-inclusive functional description of the system in our hands in the form of the source code, which determines unambiguously the physical operations carried out by the computer. It is also a widespread conception about computer programs that the source code is "nothing more" than the "shorthand writing" of assembly code, and the task of the programmer is nothing else than to reduce the problem to a symbolic functional description [2]. Overall, John's task seems to be nothing else than to do it the other way around: to trace back the variables and the functions to their declarations and initializations.

Now let us look at our example excerpt, and see what does the variable `hprd1->hsts` (functioning here as a branching condition) mean? If we run a search against the source code, we find the following initialization:

```
hprd1=&HashP1[HashRead_HM1=RGcoml[R_DM_hPtr1]];
```

This means that the value and the meaning of `hprd1` in the context of the program depend on the variables `RGcoml`, `R_DM_hPtr1` and `HashP1`. If we start to look for their meaning, we find that they are interwoven in a holistic web of relationships: they appear and get modified at many places throughout the source code. Because they are *globally declared* variables, the *temporal order* of these modifications at runtime cannot be reconstructed based on the formal declarations of the functions, within which they get modified. In order to trace back the state-changes of these variables, the temporal order of nearly *all* instances of their modifications have to be reconstructed. Even the simple task of enumerating the values they take up during the course of execution – which is indispensable to find out the states they represent – involves a lot of reconstruction and/or online debugging. This difficulty is multiplied by the various "temporal self-referentialities" that exist in the source: in our example, the variable `hprd1` appears in the branching condition `switch(hprd1->hsts)`, which influences its subsequent modification `hprd1=&HashP1[HashRead_HM1=0]`.

We can see that the problem of these holistic interrelations is that they result in a situation, where in order to understand one segment of the source code, we have to understand a great deal of it. *Understanding a single symbol presupposes understanding the whole system, but we only have at hand a description of the system that consists of such symbols.*

The interpretation moves in almost Derridean depths: those that are signified by the signs are not present; every symbol is only a deferral, a trace that leads to other symbols. In this infinite drift of semiosis, there is no fixed point to which we could anchor our interpretation [8]. We might even arrive at a "grounding" definition like this one:

```
int32*RGcoml; // Channel address (int32*)
```

but it does not help much, since we do not know what does the word "channel" mean within the holistic context of the system. (It seems to have some very situated and idiosyncratic meaning.) This definition is not grounding, but just another indication, this time pointing forward, toward the web of subsequent modifications and operations carried out on the variable.

Our example boils down to the central question: *what does it mean to understand a functional description?* It turns out that *understanding* a functional description is very different from *possessing* a fully explicit symbolic description. A 4-megabyte

mented with preprocessor macros (#include, #ifdef)

[4] The DBM runs on a few different servers.

```
#if defined(MULTI_HASH)
void*HashManager_HM1(void*arg)
{
  if(arg){}     // gcc warning
  int32 onwrk;
  Trace(8,"Start HASH comm");
  while(ServerStatus!=1){
    onwrk=1;
    do{
      switch(hprd1->hsts){
      case 2:   // Active data, update
        if(hprd1->htbl!=HM_MEMBERID)*hprd1->hptr=hprd1->hval;else{
          if(*hprd1->hptr==0)*hprd1->hptr=hprd1->hval;
                // Member ID: the first data must be recorded only
           }
        HashReadCnt_HM1++;HashLoad_HM1[hprd1->hidx]++;hprd1->hsts=0;
        if(++HashRead_HM1==HM_PIPESIZ)hprd1=&HashP1[HashRead_HM1=0];else hprd1++;
        RGcoml[R_DM_hPtr1]=HashRead_HM1;
        break;
      case 3:   // Deleted data, no update
        HashReadCnt_HM1++;HashLoad_HM1[hprd1->hidx]++;hprd1->hsts=0;
        if(++HashRead_HM1==HM_PIPESIZ)hprd1=&HashP1[HashRead_HM1=0];else hprd1++;
        RGcoml[R_DM_hPtr1]=HashRead_HM1;
        break;
      default:onwrk=0;break;
      }
    }while(onwrk);
    sleepm(2);
  }
  Trace(8,"Stop HASH comm");
  while(hprd1->hsts==2){hprd1->hsts=0;
  if(++HashRead_HM1==HM_PIPESIZ)hprd1=&HashP1[HashRead_HM1=0];else hprd1++;}
  RGcoml[R_DM_hPtr1]=HashRead_HM1;
  Trace(8,"End HASH comm");
  HashManThrJoin_HM1=1;
  return(0);
}
#endif
```

Fig. 1. Example: one of the main functions of the DBM

long source code that lacks abstraction is almost as meaningless as the original problem was before it was ever programmed. It might be easier to rewrite this code from scratch than to understand it.

At this point, I would like to recall our thesis a), according to which understanding unfolds itself in *skillful action*, and that is a certain capability of orienting ourselves in a lifeworld. In our case, John wants to know what happens if he modifies the source code: what other code regions, what functions will be affected? Where should he look for the locus of errors, when errors crop up thereafter? If he is asked to add a new feature, where shall he start to carry it out and what code regions could he build upon?

A good source code is indeed more than a shorthand writing of assembly code, insofar as it aids this kind of orientation, because the symbolic description does not always provide the reader with unambiguous points of orientation. *Moreover, in order to understand a source code line, John does not only have to know its role in the narrow technical context of the system, but he has to know also the role it plays in the holistic context of the system's prospected uses, in the future lifeworld of its users.* This is not a function-attribution carried out at an abstract, conceptual level, but a skill of orientation within the space of the projected use-situations. In order to understand the variable hprd1->hsts, we also need to be able to trace it back to users' requirements, and user interface features. This is the orientation skill Anthony can not put into words and can not convey to John. This ineffability is most probably connected to the lack of abstractions, because the linguistic articulation would necessarily involve the grouping together of various cases of lifeworld situations. The paradox we face is that *in this case, the symbolic functional description – despite that it is fully explicit – is only an imprint of an undecipherable private language*.

How can anyone orient himself in such a great jumble of source code? The lack of abstraction forces Anthony to use special methods. He customizes his development environment with sophisticated colouring schemes, and he uses colour pencils on his innumerable notes and sketches. He arranges the repeating patterns of the source code in visually prominent patterns. His horizon of understanding seems to consist of refined skills of orientation within this individualistic similarity space, which John does not share, and can not even imagine how it could be approached, because there is no common ground that they could both rely on, and Anthony's private skills resist linguistic articulation. The case is similar to Anthony's natural-language explanations and "specifications": these are just as unintelligible as the source code because they also draw upon a background of skills and knowledge that is not shared by John. The circles of interpretational efforts between them all result in frustrating failures, the convergence toward a shared horizon, appraised by Gadamer as the "miracle of understanding", does not start to

evolve. This cannot be taken as a fault of the software, but rather as an *acute crisis of understanding* within the socio-technical lifeworld of Anthony and John.

3.3 Avoiding the Crisis of Understanding through Abstraction, Embedding in a Tradition, and the Extension of Controlled Microworlds

The central problem in our example seems to be that Anthony and John approach the source code under different horizons of understanding: in their skillful interaction with the system, they take different marks to be significant for their orientation; in their abstractions, they group together use-situations by different dimensions, and so on. How does this relate to the two other principles mentioned in the introduction: embeddedness in a tradition, and the extension of controlled microworlds?

John's relation to his own horizon is quite different from that of Anthony. John is able to embed and recontextualize his abstractions in the language of traditional approaches. He is able to trace back and reduce his concepts to standard textbook terms, well-known theories, and everyday examples. When he faces lack of comprehension from his peer, because he employs unarticulated skills and tacit knowledge, he can point to books, tutorials, and paradigmatic examples, from which these non-public skills can be learned. He can also refer back to the standard background of university curricula, so that he can build up a shared orientation with his peers even in special cases when there is no preexisting, shared set of concepts among them. When he labels his variables in his source code, makes his functional divisions, and introduces abstractions in forms of functions and classes, he draws on this public and reconstructable horizon, and he even gives hints for the reconstruction in his comments. To sum up: his functional descriptions and symbolic articulations are carried out within standardized backgrounds; they are embedded into shared traditions of understanding. Whoever shares these traditions can easily understand him.

John would expect Anthony to come up with such traditionally embedded explanations, but Anthony was socialized in the pre-internet age, he built up his abstractions in his individual way, and he does not have connections to these common traditions. He is suspicious against all abstractions that are not his own. His source code is private and isolated.

John's proposed solution for the DBM would be to use an intermediate formal language, which would make it possible to formulate the problem in a traditional language – for example, by drawing on functional programming idioms or standard DBM abstractions –, but in a way that can be later processed automatically. His preferred solution would be something along the lines of *template metaprogramming* [27], which harnesses the power of the C++ compiler to define *sublanguages* of C++ that resemble traditional mathematical models or natural-language descriptions; but at the same time, can be compiled and run as any other program.

The good source code surpasses the stage of being a "shorthand writing" of assembly code because its creators and interpreters have embedded it in the horizon of various interpretative traditions.

Anthony's general strategy, "giving unique solutions for unique cases", have a further drastic influence on the possibility of grasping generalities and building abstractions. The extreme size of the source code can be attributed to the fact that all unique solutions involve a duplication of the source code, and then the slight alteration of one copy. In this DBM system, this goes so far that we cannot even talk about "sorting" in general, since there are some fifty slightly different "quicksort" routines, which are all built on different background assumptions and all induce different patterns of use. As a consequence of this, any bug fix or any alteration of the source code draws upon a highly local, system-specific, and situated knowledge; about the particular system on one side, and about the particular use-cases on the other. This situatedness makes the software hard to adapt to a new system architecture or a new user.

If we recall Rouse's model, the key to the success of the natural scientist or the technologist lies in whether he can extend his controlled microworld into the lifeworld of his users. Anthony's strategy tries to achieve this by total adaptation to each user's unique needs. But just what does "controlled" refer to in our case? A software firm does not have a laboratory, in which the experiments can be carried out under a controlled environment, in order to be reproduced! A software firm, on the other side, is a socio-technical system that converts various resources – e.g. human thinking – into software products by producing, using, and transforming source code. In order to be successful on the long run, the firm must stabilize its own code-producing processes. Anthony's idiosyncratic solutions to extend the microworld have a disastrous effect on the stability of this microworld itself: since he is a bottleneck and a critical risk factor, the microworld cannot be extended any further, it cannot be *scaled up* to serve new users and to offer new features. The successes of development are not reproducible, they are bound by the local conditions of the existing users and architectures. And a software within the global market that is bound by such internal constraints of growth is predictable to disappear among its competitors.

Extending the microworld of the software into the lifeworld of its users; but at the same time not breaking the stability of the code-producing processes is a result of a delicate balance. It involves building abstractions, grouping together use-situations and imposing the effects of such development decisions on the users when possible, but it also involves offering unique solutions to unique clients wherever it is profitable and does not risk the long-term internal stability of the firm's processes. Both extremes of the spectrum: extreme rigidity and extreme flexibility in face of possible future use-situations are risky and unstable strategies.

There are other software development methodologies, for example, eXtreme Programming [1] , which try to avoid such

traps. They emphasize peer review of code ("pair programming"), and a distributed skills and understanding of the source among programmers in the organization ("community ownership of code"). In such a methodology, Anthony would have been forced from the beginning to share his horizon of understanding and his language with his colleagues, and this might have influenced him toward using higher abstractions and embedding his concepts into the language of some shared tradition.

The relation between the three core concepts is now visible: the value of abstraction and embeddedness in an interpretative tradition are both to be measured in light of the goal of extending the microworld. Abstractions have to be carried out along dimensions that allow for maximizing the covering of the use-situations, while keeping the source code understandable. Traditions have to be followed as far as they do not constrain the perspective of understanding to a conservative tunnel vision. Making good decisions in these matters can only result from a skillful process of interpretation within the context of the lifeworld of the users and the developers.

3.4 Conclusion drawn from the case study

What is the relation between this case study and the five theses mentioned in the introduction? We have analysed the kind of knowledge that is necessary to understand the source code, and we have found that (a) it is better grasped as an implicit skill of orientation, as an ability to find ourselves around in the source code, than an explicit remembrance of the symbols. We have seen that (b) understanding a symbol is situated as far as it involves envisaging the particular system context and the lifeworld situation in which it gets used. This situatedness poses problems for the stability of the code-producing processes, which can be overcome by employing certain strategies, e.g. by standardizing and reducing the multiplicity of the use-situations through abstraction, or by leaning on and adapting a previously given tradition of understanding (c). Our case study also shed light on the interpretative processes – characteristic of software development – aimed at understanding the source code and the lifeworld use-situations as well (d).

4 General conclusions

In this essay, I argued for a revised understanding of software development as social activity. I have drawn upon the insights of Distributed Cognition and its theoretical roots, but I have tried to shift the emphasis from the classical cognitivist themes to the topics of situatedness, embodiment, skillful action, interpretation, and tradition, which are characteristic of the post-classical cognitivism of Brooks, Suchman, Winograd, Dreyfus, Maturana, Lakoff and Norman. I articulated my theses in the introduction in four points, and demonstrated them in a case study. I argued that the case study exemplifies a situation that is typical enough, so that the insights gained can be generalized to the whole field of information technologies.

References

1 **Beck K**, *Extreme programming explained: Embrace change*, Addison-Wesley Professional, 1999.

2 **Binzberger V**, *A szoftverhiba jelensége hermeneutikai megközelítésből*, Világosság **2006**, no. 3, 27-34.

3 **Boehm B, Turner R**, *Balancing Agility and Discipline*, Addison-Wesley, 2004.

4 **Brooks R A**, *Intelligence without representation*, Artificial Intelligence **47** (1991), 139–159.

5 **Capurro R**, *Informatics and Hermeneutics* (Floyd, Züllighoven, Budde, Keil-Slawik, eds.), Springer-Verlag, New York, 1992.

6 **Clark A**, *Being There*, MIT Press, 1997.

7 **Demarco T, Lister T**, *Peopleware*, 2nd ed., Dorset House Publishing, New York, 1999.

8 **Derrida J**, *Of Grammatology*, John Hopkins University Press, Baltimore, 1997.

9 **Dittrich Y, Christiane F, Ralf K**, *Social Thinking - Software Practice*, MIT Press, Cambridge, 2002.

10 **Dourish P**, *Where The Action Is: The Foundations of Embodied Interaction*, MIT Press, 2001.

11 **Dreyfus HL**, *Why we do not have to worry about speaking the language of the computer*, Information Technology & People **11** (1998), no. 4, 281-289.

12 **Gadamer HG**, *Truth and Method*, translated by Weinsheimer J, Marshall D G, Crossroad, New York, 1989.

13 **Galison P**, *Computer Simulations*, The Disunity of Science, 1996, pp. 118-157.

14 **Halverson C**, *Distributed cognition as a theoretical framework for HCI: Do Not throw the baby out with the bathwater – the importance of the cursor in air traffic control*.

15 **Heelan PA**, *Yes! There Is a Hermeneutics of Natural Science: A Rejoinder to Markus*, Science in Context **3** (1989), no. 2, 477-488.

16 **Heelan PA, Schulkin J**, *Hermeneutical Philosophy and Pragmatism: A Philosophy of Science, Philosophy of Technology: An Anthology* (Scharff RC, Dusek V, eds.), Blackwell Publishing, 2003.

17 **Heidegger M**, *Being and Time*, translated by Joan Stambaugh, State University of New York Press, 1996.

18 ———, *Das Ende der Philosophie und die Aufgabe des Denkens*, 2002. Vortrag GA14. Vittorio Klostermann.

19 **Hollan J, Edwin H, David K**, *Distributed Cognition: Toward a New Foundation for Human-Computer Interaction Research*, ACM Transactions on Computer-Human Interaction **7** (2000), no. 2, 174–196.

20 **Hounshell D**, *Du Pont and Large-Scale R&D*, Big Science: The Growth of Large-Scale Research, Hevly B, Stanford, CA: Stanford University Press, 1992.

21 **Hunt A, Thomas D**, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley, 1999.

22 **Hutchins E, Palen L**, *Constructing Meaning from Space, Gesture and Speech*, Discourse, Tools, and Reasoning: Essays on Situated Cognition, Burge B, Springer-Verlag, Heidelberg Germany, 1997, pp. 23-40.

23 **Hutchins E**, *Cognition in the Wild*, MIT Press, Cambridge MA, 1994.

24 ———, *How the cockpit remembers its speed*, Cognitive Science **19** (1995), 265-288.

25 **Ihde D**, *A Phenomenology of Technics*, Philosophy of Technology (Scharff R C, Dusek V, eds.), Blackwell Publishing, Oxford, 2003.

26 **John MC (ed.)**, *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, Morgan Kaufmann, 2003.

27 **Karlsson B**, *Beyond the C++ Standard Library: An Introduction to Boost*, Addison Wesley Professional, 2005.

28 **Kisiel TJ**, *Heidegger és az új tudománykép, Hermeneutika és a természettudományok* (Margitay T, Schwendtner T, eds.), Áron kiadó, Budapest, 2001.

29 **Lakoff G, Johnson M**, *Metaphors we live by*, University of Chicago Press, Chicago, 1980.

30 **Márkus G**, *Why is There No Hermeneutics of Natural Science? Some Preliminary Theses*, Science in Context **1** (1987), 5-51.

31 **Maturana H, Varela F**, *Autopoesis and Cognition. A Realization of the Living*, Reidel Publishing Company, Dordrecht, 1980.

32 **Mcbreen P**, *Software Craftmanship - The New Imperative*, Addison – Wesley, 2002.

33 **Mitcham C**, *Introduction. The Blackwell Guide to the Philosophy of Computing and Information* (Floridi L, ed.), Blackwell Publishing, Oxford, 2004.

34 **Newell A, Simon H**, *Computer Science as Empirical Inquiry: Symbols and Search, Communications of the ACM* **19** (1976), no. 3.

35 **Norman DA**, *The Design of Everyday Things*, The MIT Press, 1998.

36 **Ramberg B, Gjesdal K**, *Hermeneutics. Entry in the Stanford Encyclopedia of Philosophy*, 2005, available at `http://plato.stanford.edu/entries/hermeneutics/`.

37 **Rouse J**, *Knowledge and Power: Toward a Political Philosophy of Science*, Cornell University Press, 1987.

38 _____, *Engaging Science: How to Understand Its Practices Philosophically*, Cornell University Press, 1996.

39 **Suchman L**, *Plans and Situated Actions: The Problem of Human-Machine Communication*, Cambridge Univ. Press, Cambridge, England, 1987.

40 **Susi T**, *The Puzzle Of Social Activity The Significance Of Tools In Cognition And Cooperation*, 2006.

41 **Tomasello M, Carpenter M, Call J, Behne T, Moll H**, *Understanding and sharing intentions: The origins of cultural cognition*, Behavioral And Brain Sciences **28** (2005), 675–735.

42 **Torres-Gregory W**, *Heidegger On Traditional Language And Technological Language*, 1998.

43 **Winograd T, Flores F**, *Understanding Computers and Cognition*, Norwood, NJ: Ablex Corporation, 1987.